First Experiences with Porting a Multigrid Solver to the CBEA

Matthias Bolten

<m.bolten@fz-juelich.de>

Central Institute for Applied Mathematics
Research Centre Jülich
Germany

ScicomP13
Computer Center Garching (RZG)
2007/07/19

Outline

Outline

Ingredients

- Cell processor consists of:

Ingredients

- Cell processor consists of:
  - 1 PowerPC Processor Element (PPE)
    General-purpose 64-bit RISC processor conforming to PowerPC Architecture (compatible to e.g. PowerPC 970).

Ingredients

- Cell processor consists of:
  - 1 PowerPC Processor Element (PPE)
    General-purpose 64-bit RISC processor conforming to PowerPC Architecture (compatible to e.g. PowerPC 970).
  - 8 Synergistic Processor Elements (SPE)
    Special purpose 128-bit RISC processor for SIMD applications.

Ingredients

- Cell processor consists of:
  - 1 PowerPC Processor Element (PPE)
    General-purpose 64-bit RISC processor conforming to PowerPC Architecture (compatible to e.g. PowerPC 970).
  - 8 Synergistic Processor Elements (SPE)
    Special purpose 128-bit RISC processor for SIMD applications.
- Typically runs at 3.2 GHz, yielding >200 GFlops in single precison and >20 GFlops in double precision

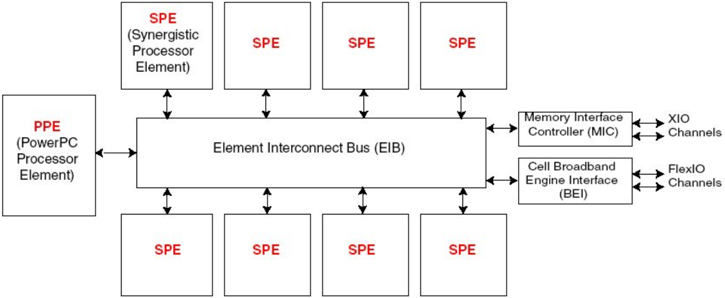
Ingredients

- Cell processor consists of:
  - 1 PowerPC Processor Element (PPE)
    General-purpose 64-bit RISC processor conforming to PowerPC Architecture (compatible to e.g. PowerPC 970).
  - 8 Synergistic Processor Elements (SPE)
    Special purpose 128-bit RISC processor for SIMD applications.
- Typically runs at 3.2 GHz, yielding >200 GFlops in single precison and >20 GFlops in double precision
- Memory bandwidth is up to 22.4 GB/s

Ingredients

- Cell processor consists of:
  - 1 PowerPC Processor Element (PPE)
    General-purpose 64-bit RISC processor conforming to PowerPC Architecture (compatible to e.g. PowerPC 970).
  - 8 Synergistic Processor Elements (SPE)
    Special purpose 128-bit RISC processor for SIMD applications.
- Typically runs at 3.2 GHz, yielding >200 GFlops in single precison and >20 GFlops in double precision
- Memory bandwidth is up to 22.4 GB/s
- It has been designed by the Cell consortium (IBM, Sony and Toshiba) and is used in the Playstation 3

Architecture



PPE

- Full set of 64-bit PowerPC registers

PPE

- Full set of 64-bit PowerPC registers
- In order execution

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache
- 32-KB level 1 data cache

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache
- 32-KB level 1 data cache
- Memory management unit

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache
- 32-KB level 1 data cache
- Memory management unit
- Power Processor Unit supports two simultaneous threads

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache
- 32-KB level 1 data cache
- Memory management unit
- Power Processor Unit supports two simultaneous threads
- 512-KB level 2 cache in Power Processor Storage Subsystem

PPE

- Full set of 64-bit PowerPC registers
- In order execution
- IEEE 754-compliant
- 32 128-bit vector registers (Altivec instruction set)
- 32-KB level 1 instruction cache
- 32-KB level 1 data cache
- Memory management unit
- Power Processor Unit supports two simultaneous threads
- 512-KB level 2 cache in Power Processor Storage Subsystem
- PPSS can communicate with the SPEs via Element Interconnect Bus (EIB)

SPE

- Not IEEE 754-compliant

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB
- Only the MFC can move data from main storage to LS and communicate with other SPEs and with the PPE

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB
- Only the MFC can move data from main storage to LS and communicate with other SPEs and with the PPE
- Data is transfered using DMA transfers, so overlapping of load/store and computation is possible

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB
- Only the MFC can move data from main storage to LS and communicate with other SPEs and with the PPE
- Data is transfered using DMA transfers, so overlapping of load/store and computation is possible
- DMA transfer list of up to 2048 transfers, each of size up to 16 KB

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB
- Only the MFC can move data from main storage to LS and communicate with other SPEs and with the PPE
- Data is transfered using DMA transfers, so overlapping of load/store and computation is possible
- DMA transfer list of up to 2048 transfers, each of size up to 16 KB
- Transfer requests can be scheduled by this SPU, by another SPE or by the PPE

SPE

- Not IEEE 754-compliant
- Consists of Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller
- SPU has large register file with 128 128-bit registers (2 KB registers)
- LS is intended for instructions and data and has a size of 256 KB
- Only the MFC can move data from main storage to LS and communicate with other SPEs and with the PPE
- Data is transfered using DMA transfers, so overlapping of load/store and computation is possible
- DMA transfer list of up to 2048 transfers, each of size up to 16 KB
- Transfer requests can be scheduled by this SPU, by another SPE or by the PPE
- Mailbox and signaling mechanisms available as well

Outline

JUICe - Juelich Initiative Cell Cluster

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux
- 1 Login node IBM System x3550 with 2 CPUs

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux
- 1 Login node IBM System x3550 with 2 CPUs
  - Processortype: dual core Intel Xeon 3.0 GHz

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux
- 1 Login node IBM System x3550 with 2 CPUs
  - Processortype: dual core Intel Xeon 3.0 GHz
  - Main memory: 2 Gbytes

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux
- 1 Login node IBM System x3550 with 2 CPUs
  - Processortype: dual core Intel Xeon 3.0 GHz
  - Main memory: 2 Gbytes
  - Disk capacity for user data: 300 GB

JUICe - Juelich Initiative Cell Cluster

- 2 BladeCenter QS20 chassis, 6 cell blades each (total 12)
  - Cell blade: 2 x cell BE processors 3.2 GHz
  - Main memory: 12 x 2 x 512 MB (aggregate 12 GB)
  - Network: Infiniband 4x / Voltaire switch ISR 9024
  - Operating system: Fedora Core-5 based Linux
- 1 Login node IBM System x3550 with 2 CPUs
  - Processortype: dual core Intel Xeon 3.0 GHz
  - Main memory: 2 Gbytes
  - Disk capacity for user data: 300 GB
  - Operating system: SuSE Linux 10.1

Outline

Programming models

As mentioned, the Cell architecture allows communication in different ways:

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

- Distribution of the data to the various SPEs

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

- Distribution of the data to the various SPEs
- Pipelining (e.g. for audio-/video-decoding, processing and reencoding)

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

- Distribution of the data to the various SPEs
- Pipelining (e.g. for audio-/video-decoding, processing and reencoding)
- Offloading some tasks from the PPE to the SPE dynamically

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

- Distribution of the data to the various SPEs
- Pipelining (e.g. for audio-/video-decoding, processing and reencoding)
- Offloading some tasks from the PPE to the SPE dynamically
- ...

Programming models

As mentioned, the Cell architecture allows communication in different ways:

- Shared memory
- Direct copy from local store to local store
- Mailboxes
- SPE to SPE signaling

That allows the use of a variety of programming models:

- Distribution of the data to the various SPEs
- Pipelining (e.g. for audio-/video-decoding, processing and reencoding)
- Offloading some tasks from the PPE to the SPE dynamically
- ...

Besides the internal on-chip parallelism, various Cell chips can be used via MPI.

Outline

Porting software to the Cell architecture

Assuming that a data distribution approach is chosen, the porting of new software consists of three steps:

Porting software to the Cell architecture

Assuming that a data distribution approach is chosen, the porting of new software consists of three steps:

1. Write the programm in C as usual

Porting software to the Cell architecture

Assuming that a data distribution approach is chosen, the porting of new software consists of three steps:

1. Write the programm in C as usual
2. Vectorizing the code using Altivec-intrinsics on the PPE

Porting software to the Cell architecture

Assuming that a data distribution approach is chosen, the porting of new software consists of three steps:

1. Write the programm in C as usual
2. Vectorizing the code using Altivec-intrinsics on the PPE
3. Replace Altivec-intrinsics by SPU-intrinsics and implement copying from and to local store on the SPE (Multi-buffering is necessary, here)

Outline

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error
- After a few pre-smoothing iterations residual $\mathbf{r}_l^{(i)} = \mathbf{b}_l - \mathbf{A}_l \mathbf{x}_l^{(i)}$ is transfered to coarser grid $l + 1$

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error
- After a few pre-smoothing iterations residual $\mathbf{r}_l^{(i)} = \mathbf{b}_l - \mathbf{A}_l \mathbf{x}_l^{(i)}$ is transfered to coarser grid $l + 1$
- Correction $\mathbf{c}_{l+1}^{(i)} = \mathbf{A}_{l+1} \mathbf{r}_{l+1}^{(i)}$ is computed using recursive application

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error
- After a few pre-smoothing iterations residual $\mathbf{r}_l^{(i)} = \mathbf{b}_l - \mathbf{A}_l\mathbf{x}_l^{(i)}$ is transfered to coarser grid $l+1$
- Correction $\mathbf{c}_{l+1}^{(i)} = \mathbf{A}_{l+1}\mathbf{r}_{l+1}^{(i)}$ is computed using recursive application
- $\mathbf{c}_{l+1}^{(i)}$ is transfered back to grid l, $\mathbf{x}_l^{(i)}$ is corrected and post-smoothed

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error
- After a few pre-smoothing iterations residual $\mathbf{r}_l^{(i)} = \mathbf{b}_l - \mathbf{A}_l\mathbf{x}_l^{(i)}$ is transfered to coarser grid $l+1$
- Correction $\mathbf{c}_{l+1}^{(i)} = \mathbf{A}_{l+1}\mathbf{r}_{l+1}^{(i)}$ is computed using recursive application
- $\mathbf{c}_{l+1}^{(i)}$ is transfered back to grid l, $\mathbf{x}_l^{(i)}$ is corrected and post-smoothed
- Each of the involved $\mathbf{A}_l$ is appropriate discretization of underlying continous problem

Multigrid methods

- Optimal iterative solvers for large class of problems (elliptic PDEs)
- Expoit the fact that iterative methods like Jacobi smooth error
- After a few pre-smoothing iterations residual $\mathbf{r}_l^{(i)} = \mathbf{b}_l - \mathbf{A}_l\mathbf{x}_l^{(i)}$ is transfered to coarser grid $l + 1$
- Correction $\mathbf{c}_{l+1}^{(i)} = \mathbf{A}_{l+1}\mathbf{r}_{l+1}^{(i)}$ is computed using recursive application
- $\mathbf{c}_{l+1}^{(i)}$ is transfered back to grid l, $\mathbf{x}_l^{(i)}$ is corrected and post-smoothed
- Each of the involved $\mathbf{A}_l$ is appropriate discretization of underlying continous problem
- Extension of approach for more general problems possible (algebraic multigrid)

Outline

Preliminary thoughts

- Most of the operations occur in smoother

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"
- Smoother has bad computation to load/store ratio (1 multplikation and 6 additions for 2 loads and one store for 3D-Laplacian)

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"
- Smoother has bad computation to load/store ratio (1 multplikation and 6 additions for 2 loads and one store for 3D-Laplacian)
- Structure can be easily exploited on the Cell architecture, think of LS as software managed cache

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"
- Smoother has bad computation to load/store ratio (1 multplikation and 6 additions for 2 loads and one store for 3D-Laplacian)
- Structure can be easily exploited on the Cell architecture, think of LS as software managed cache
- For d-dimensional nearest neighbur problem only 3 $(d-1)$-dimensional slices have to be available locally

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"
- Smoother has bad computation to load/store ratio (1 multplikation and 6 additions for 2 loads and one store for 3D-Laplacian)
- Structure can be easily exploited on the Cell architecture, think of LS as software managed cache
- For d-dimensional nearest neighbur problem only 3 $(d-1)$-dimensional slices have to be available locally
- Data is ditributed amongst SPEs in the $(d-1)$th dimension to load large consecutive chunks from global main memory

Preliminary thoughts

- Most of the operations occur in smoother
- As a consequence, smoother was chosen as "porting prototype"
- Smoother has bad computation to load/store ratio (1 multplikation and 6 additions for 2 loads and one store for 3D-Laplacian)
- Structure can be easily exploited on the Cell architecture, think of LS as software managed cache
- For d-dimensional nearest neighbur problem only 3 $(d-1)$-dimensional slices have to be available locally
- Data is ditributed amongst SPEs in the $(d-1)$th dimension to load large consecutive chunks from global main memory
- Pipelining well-suited to improve computation to load/store ratio (chaining smoothing iterations)

Step 1: Plain C program

```
float u[N+2][N+2][N+2];
float f[N+2][N+2][N+2];
float tmp[N+2][N+2][N+2];
...
float f1_6 = 1.0/6.0;
...
for (iter=0;iter<MAXITER;iter++) {
  for (i=1;i<N+1;i++) {
    for (j=1;j<N+1;j++) {
      for (k=1;k<N+1;k++) {
        tmp[i][j][k] = f1_6 * (u[i][j][k-1] + u[i][j][k+1] +
                               u[i][j-1][k] + u[i][j+1][k] +
                               u[i-1][j][k] + u[i+1][j][k] +
                               f[i][j][k]);
      }
    }
  }
}
```

Step 2: C program plus Altivec intrinsics

```
vector float u[N/4+2][N+2][N+2];
vector float f[N/4+2][N+2][N+2];
vector float tmp[N/4+2][N+2][N+2];
...
vector float zero, sl, sr, fac;
...
for (iter=0;iter<MAXITER;iter++) {
    for (i=1;i<N/4+1;i++) {
        for (j=1;j<N+1;j++) {
            for (k=1;k<N+1;k++) {
               tmp[i][j][k] = f[i][j][k];
               sl = vec_sld(u[i][j][k],u[i+1][j][k],4);
               sr = vec_sld(u[i-1][j][k],u[i][j][k],12);
               tmp[i][j][k] = vec_add(tmp[i][j][k],u[i][j-1][k]);
               tmp[i][j][k] = vec_add(tmp[i][j][k],u[i][j+1][k]);
               tmp[i][j][k] = vec_add(tmp[i][j][k],u[i][j][k-1]);
               tmp[i][j][k] = vec_add(tmp[i][j][k],u[i][j][k+1]);
               tmp[i][j][k] = vec_madd(fac,tmp[i][j][k],zero);
}   }   }   }
```

Step 3: SPU C program (1/3)

```
...
copy_data(my_u[0],u+(m+2)*(       ystart),(m+2)*(yend-ystart+2),
          bufidx,MFC_GET_CMD);
copy_data(my_u[1],u+(m+2)*(   n+2+ystart),(m+2)*(yend-ystart+2),
          bufidx,MFC_GET_CMD);
copy_data(my_u[2],u+(m+2)*(2*n+4+ystart),(m+2)*(yend-ystart+2),
          bufidx,MFC_GET_CMD);
copy_data(my_f[0],f+(m+2)*(       ystart),(m+2)*(yend-ystart+2),
          bufidx,MFC_GET_CMD);

/* Jacobi iteration */
for (int k=1; k<=o; k++) {
  nextidx = bufidx ^ 1;
  /* Copying data for next plane */
  copy_data(my_u[3],u+(m+2)*((k+2)*(n+2)+ystart),
            (m+2)*(yend-ystart+2),nextidx,MFC_GET_CMD);
  copy_data(my_f[1],f+(m+2)*((k+1)*(n+2)+ystart),
            (m+2)*(yend-ystart+2),nextidx,MFC_GET_CMD);
```

Step 3: SPU C program (2/3)

```
/* Doing computation on current plane */
 for (int j=1; j<=yend-ystart; j++) {
   for (int i=1; i<=m; i++) {
    sr1 = spu_rlqwbyte(my_u[1][i  +(j  )*m],12);
    sr2 = spu_slqwbyte(my_u[1][i  +(j-1)*m],12);
    sr = spu_madd(sr1,last_3,sr2);
    sl1 = spu_rlqwbyte(my_u[1][i  +(j+1)*m],4);
    sl2 = spu_slqwbyte(my_u[1][i  +(j  )*m],4);
    sl = spu_madd(sl1,last_1,sl2);

    tmp1 = spu_add(sl,sr);
    tmp2 = spu_add(my_f[0][i+j*m],tmp1);
    tmp3 = spu_add(my_u[1][i-1+(j  )*m],my_u[1][i+1+(j  )*m]);
    tmp4 = spu_add(my_u[0][i  +(j  )*m],my_u[2][i  +(j  )*m]);
    tmp5 = spu_add(tmp3,tmp4);
    tmp6 = spu_add(tmp2,tmp5);
    my_tmp[0][i+j*m] = spu_mul(fac,tmp6);
   }
 }
```

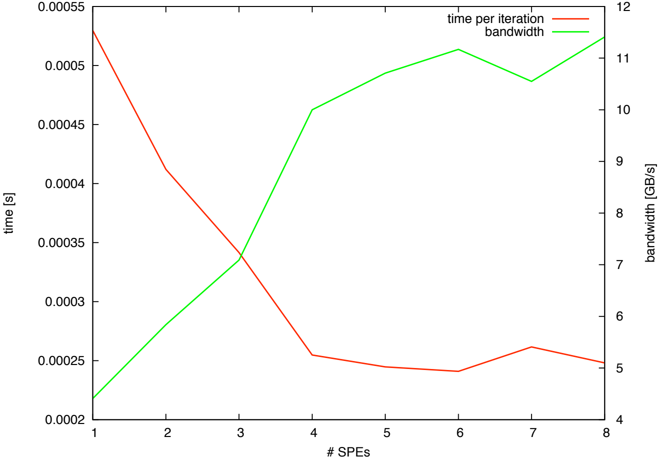
Step 3: SPU C program (3/3)

```
/* Swapping pointers */
tmp_ptr = my_u[0];
my_u[0] = my_u[1];
my_u[1] = my_u[2];
my_u[2] = my_u[3];
my_u[3] = tmp_ptr;
tmp_ptr = my_f[0];
my_f[0] = my_f[1];
my_f[1] = tmp_ptr;
tmp_ptr = my_tmp[0];
my_tmp[0] = my_tmp[1];
my_tmp[1] = tmp_ptr;

/* Copying data back */
 copy_data(my_tmp[1],u+(m+2)*(k*(n+2)+ystart),
           (m+2)*(yend-ystart+2),nextidx,MFC_PUT_CMD);
...
```

Outline

Preliminary results



Outline

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming
- The performance results are promising

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming
- The performance results are promising

What's missing?

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming
- The performance results are promising

What's missing?

- IEEE arithmetic

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming
- The performance results are promising

What's missing?

- IEEE arithmetic
- Double precision

Conclusion

Lessons learned:

- The Cell Broadband Engine Architecture allows a wide variety of programming models
- Use of these models is supported by hardware mechanisms
- Porting of existing software is time-consuming
- The performance results are promising

What's missing?

- IEEE arithmetic
- Double precision
- Tools for porting software

First Experiences with Porting a Multigrid Solver to the CBEA

Matthias Bolten

<m.bolten@fz-juelich.de>

Central Institute for Applied Mathematics
Research Centre Jülich
Germany

SciComP13
Computer Center Garching (RZG)
2007/07/19